MULTITHREADED PROGRAMMING

- In a *thread-based* multitasking environment, the thread is the smallest unit of dispatchable code
- Multitasking threads require less overhead than multitasking processes
- Processes are heavyweight tasks that require their own separate address spaces
- Interprocess communication is expensive and limited
- Context switching from one process to another is also costly
- Threads, on the other hand, are lightweight
- They share the same address space and cooperatively share the same heavyweight process
- Interthread communication is inexpensive, and context switching from one thread to the next is low cost
- Multithreading enables you to write very efficient programs that make maximum use of the CPU, because idle time can be kept to a minimum

The Java Thread Model

- Single-threaded systems use an approach called an *event loop* with *polling*
- In this model, a single thread of control runs in an infinite loop, polling a single event queue to decide what to do next
- Once this polling mechanism returns with, say, a signal that a network file is ready to be read, then the event loop dispatches control to the appropriate event handler
- Until this event handler returns, nothing else can happen in the system
- This wastes CPU time
- In general, in a singled-threaded environment, when a thread *blocks* (that is, suspends execution) because it is waiting for some resource, the entire program stops running
- When a thread blocks in a Java program, only the single thread that is blocked pauses; All other threads continue to run.
- Threads exist in several states
- A thread can be *running*
- It can be *ready to run* as soon as it gets CPU time
- A running thread can be *suspended,* which temporarily suspends its activity
- A suspended thread can then be *resumed,* allowing it to pick up where it left off
- A thread can be *blocked* when waiting for a resource
- At any time, a thread can be terminated, which halts its execution immediately. Once terminated, a thread cannot be resumed

Thread Priorities

- Java assigns to each thread a priority that determines how that thread should be treated with respect to the others
- Thread priorities are integers that specify the relative priority of one thread to another

- A thread's priority is used to decide when to switch from one running thread to the next

- This is called a *context switch*
- The rules that determine when a context switch takes place are:
- *A thread can voluntarily relinquish control*
- *A thread can be preempted by a higher-priority thread*

Synchronization

- Java implements synchronization using ***monitor***
- Once a thread enters a monitor, all other threads must wait until that thread exits the monitor
- Once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

Messaging

- Java provides a clean, low-cost way for two or more threads to talk to each other, via calls to predefined methods that all objects have
- Java's messaging system allows a thread to enter a synchronized method on an object, and then wait there until some other thread explicitly notifies it to come out

The Thread Class and the Runnable Interface

- Java's multithreading system is built upon the **Thread** class, its methods, and its companion interface, **Runnable**
- To create a new thread, our program will either extend **Thread** or implement the **Runnable** interface

The Main Thread

- When a Java program starts up, one thread begins running immediately
- This is usually called the *main thread* of your program, because it is the one that is executed when your program begins
- It is the thread from which other "child" threads will be spawned
- Often it must be the last thread to finish execution because it performs various shutdown actions
- Although the main thread is created automatically when your program is started, it can be controlled through a **Thread** object
- To do so, we must obtain a reference to it by calling the method **currentThread( )**, which is a **public static** member of **Thread**
  static Thread currentThread( )
- This method returns a reference to the thread in which it is called

```
// Controlling the main Thread.
class CurrentThreadDemo {
public static void main(String args[]) {
Thread t = Thread.currentThread();
System.out.println("Current thread: " + t);
// change the name of the thread
t.setName("My Thread");
System.out.println("After name change: " + t);
try {
for(int n = 5; n > 0; n--) {
System.out.println(n);
Thread.sleep(1000);
} } catch (InterruptedException e) {
System.out.println("Main thread interrupted");
} } }
```

- The argument to **sleep( )** specifies the delay period in milliseconds

<u>Output</u>:

```
Current thread: Thread[main,5,main]
After name change: Thread[My Thread,5,main]
5   4  3  2  1
```

- ***t*** displays the name of the thread, its priority, and the name of its group
- By default, the name of the main thread is **main**.
- Its priority is 5, which is the default value, and **main** is also the name of the group of threads to which this thread belongs
- A *thread group* is a data structure that controls the state of a collection of threads as a whole
- We can set the name of a thread by using **setName( )**
- We can obtain the name of a thread by calling **getName( )**

<u>Creating a Thread</u>

- We create a thread by instantiating an object of type **Thread**
- We can implement the **Runnable** interface
- We can extend the **Thread** class, itself

<u>Implementing Runnable</u>

- We can construct a thread on any object that implements **Runnable**
- To implement **Runnable**, a class need only implement a single method called **run( )**, which is declared like this:

  public void run( )
- Inside **run( )**, you will define the code that constitutes the new thread
- Inside **run( )**, we will define the code that constitutes the new thread
- **run( )** establishes the entry point for another, concurrent thread of execution within our program
- After we create a class that implements **Runnable**, we will instantiate an object of type **Thread** from within that class
- The constructor is:

  Thread(Runnable *threadOb*, String *threadName*) *threadOb* is an instance of a class that implements the **Runnable** interface; This defines where execution of the thread will begin

- The name of the new thread is specified by *threadName*
- After the new thread is created, it will not start running until you call its **start( )** method, which is declared within **Thread**
- In essence, **start( )** executes a call to **run( )**

```
class NewThread implements Runnable {
Thread t;
NewThread() {
// Create a new, second thread
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}  } catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}  }
class ThreadDemo {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}  } catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
} }
```

Extending Thread

- The second way to create a thread is to create a new class that extends **Thread**, and then to create an instance of that class

The extending class must override the **run( )** method, which is the entry point for the new thread

- It must also call **start( )** to begin execution of the new thread

```
// Create a second thread by extending Thread
```

```
class NewThread extends Thread {
NewThread() {
// Create a new, second thread
super("Demo Thread");
System.out.println("Child thread: " + this);
start(); // Start the thread
}
// This is the entry point for the second thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
} } catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
} }
class ExtendThread {
public static void main(String args[]) {
new NewThread(); // create a new thread
try {
for(int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
} } catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
} }
```

- Call to **super( )** inside **NewThread** invokes the following form of the **Thread** constructor: ***public Thread(String threadName)***
- Here, *threadName* specifies the name of the thread

<u>Choosing an Approach</u>

- Which approach is better?
- The **Thread** class defines several methods that can be overridden by a derived class
- Of these methods, the only one that *must* be overridden is **run( )**
- This is, of course, the same method required when you implement **Runnable**
- Many Java programmers feel that classes should be extended only when they are being enhanced or modified in some way
- So, if you will not be overriding any of **Thread**'s other methods, it is probably best simply to implement **Runnable**
- But it is left to the programmers;

<u>Creating Multiple Threads</u>

- So far, we have been using only two threads: the main thread and one child thread
- However, our program can spawn as many threads as it needs

```
// Create multiple threads.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
} } catch (InterruptedException e) {
System.out.println(name + "Interrupted");
}
System.out.println(name + " exiting.");
} }
class MultiThreadDemo {
public static void main(String args[]) {
new NewThread("One"); // start threads
new NewThread("Two");
new NewThread("Three");
try {
// wait for other threads to end
Thread.sleep(10000);
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Main thread exiting.");
}}
```

Using isAlive( ) and join( )

- How can one thread know when another thread has ended?
- Two ways exist to determine whether a thread has finished
- First, you can call ***isAlive( )*** on the thread
- This method is defined by **Thread**, and its general form is

  final boolean isAlive( )
- The **isAlive( )** method returns **true** if the thread upon which it is called is still running
- It returns **false** otherwise

- The method that you will more commonly use to wait for a thread to finish is called **join( )**, shown here:
  final void join( ) throws InterruptedException
- This method waits until the thread on which it is called terminates

```
// Using join() to wait for threads to finish.
class NewThread implements Runnable {
String name; // name of thread
Thread t;
NewThread(String threadname) {
name = threadname;
t = new Thread(this, name);
System.out.println("New thread: " + t);
t.start(); // Start the thread
}
// This is the entry point for thread.
public void run() {
try {
for(int i = 5; i > 0; i--) {
System.out.println(name + ": " + i);
Thread.sleep(1000);
} } catch (InterruptedException e) {
System.out.println(name + " interrupted.");
}
System.out.println(name + " exiting.");
} }
class DemoJoin {
public static void main(String args[]) {
NewThread ob1 = new NewThread("One");
NewThread ob2 = new NewThread("Two");
NewThread ob3 = new NewThread("Three");
System.out.println("Thread One is alive: " + ob1.t.isAlive());
System.out.println("Thread Two is alive: " + ob2.t.isAlive());
System.out.println("Thread Three is alive: " + ob3.t.isAlive());
// wait for threads to finish
try {
System.out.println("Waiting for threads to finish.");
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch (InterruptedException e) {
System.out.println("Main thread Interrupted");
}
System.out.println("Thread One is alive: " + ob1.t.isAlive());
System.out.println("Thread Two is alive: " + ob2.t.isAlive());
System.out.println("Thread Three is alive: " + ob3.t.isAlive());
System.out.println("Main thread exiting.");
} }
```

Synchronization

- When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time
- The process by which this is achieved is called *synchronization*

- Key to synchronization is the concept of the monitor (like semaphore)
- A *monitor* is an object that is used as a mutually exclusive lock, or *mutex*
- Only one thread can *own* a monitor at a given time
- When a thread acquires a lock, it is said to have *entered* the monitor
- All other threads attempting to enter the locked monitor will be suspended until the first thread *exits* the monitor
- These other threads are said to be *waiting* for the monitor

Using Synchronized Methods

```
// This program is not synchronized.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch(InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
} }
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
public void run() {
target.call(msg);
} }
class Synch {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
```

```
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
} } }
```

Here is the output produced by this program:

```
[Hello[Synchronized[World]
]
]
```

- In this program, nothing exists to stop all three threads from calling the same method, on the same object, at the same time
- This is known as a ***race condition***, because the three threads are racing each other to complete the method
- In most situations, a race condition is more subtle and less predictable, because you can't be sure when the context switch will occur
- To fix the preceding program, you must *serialize* access to **call( )**
- To do this, you simply need to precede **call()**'s definition with the keyword **synchronized**

```
class Callme {
synchronized void call(String msg) {
        ...
```

- This prevents other threads from entering **call( )** while another thread is using it
- After **synchronized** has been added to **call()**, the output of the program is as follows:

```
[Hello]
[Synchronized]
[World]
```

- Once a thread enters any synchronized method on an instance, no other thread can enter any other synchronized method on the same instance
- However, nonsynchronized methods on that instance will continue to be callable

The synchronized Statement

- Imagine that you want to synchronize access to objects of a class that was not designed for multithreaded access
- That is, the class does not use **synchronized** methods
- Further, this class was not created by you, but by a third party, and you do not have access to the source code
- Thus, you can't add **synchronized** to the appropriate methods within the class

How can access to an object of this class be synchronized?
You simply put calls to the methods defined by this class inside a synchronized block

- This is the general form of the synchronized statement:

```
synchronized(object) {
        // statements to be synchronized
}
```

- Here, *object* is a reference to the object being synchronized
- A synchronized block ensures that a call to a method that is a member of *object* occurs only after the current thread has successfully entered *object*'s monitor

```
// This program uses a synchronized block.
class Callme {
void call(String msg) {
System.out.print("[" + msg);
try {
Thread.sleep(1000);
} catch (InterruptedException e) {
System.out.println("Interrupted");
}
System.out.println("]");
} }
class Caller implements Runnable {
String msg;
Callme target;
Thread t;
public Caller(Callme targ, String s) {
target = targ;
msg = s;
t = new Thread(this);
t.start();
}
// synchronize calls to call()
public void run() {
synchronized(target) { // synchronized block
target.call(msg);
} } }
class Synch1 {
public static void main(String args[]) {
Callme target = new Callme();
Caller ob1 = new Caller(target, "Hello");
Caller ob2 = new Caller(target, "Synchronized");
Caller ob3 = new Caller(target, "World");
// wait for threads to end
try {
ob1.t.join();
ob2.t.join();
ob3.t.join();
} catch(InterruptedException e) {
System.out.println("Interrupted");
} } }
```

Interthread Communication

- Java includes an elegant interprocess communication mechanism via the **wait( )**, **notify( )**, and **notifyAll( )** methods

- All three methods can be called only from within a **synchronized** context
- **wait( )** tells the calling thread to give up the monitor and go to sleep until some other thread enters the same monitor and calls **notify( )**
- **notify( )** wakes up the first thread that called **wait( )** on the same object

- **notifyAll( )** wakes up all the threads that called **wait( )** on the same object; The highest priority thread will run first.

```
class Q {
int n;
boolean valueSet = false;
synchronized int get() {
if(!valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
System.out.println("Got: " + n);
valueSet = false;
notify();
return n;
}
synchronized void put(int n) {
if(valueSet)
try {
wait();
} catch(InterruptedException e) {
System.out.println("InterruptedException caught");
}
this.n = n;
valueSet = true;
System.out.println("Put: " + n);
notify();
} }
class Producer implements Runnable {
Q q;
Producer(Q q) {
this.q = q;
new Thread(this, "Producer").start();
}
public void run() {
int i = 0;
while(true) {
q.put(i++);
} } }
class Consumer implements Runnable {
```

```
Q q;
Consumer(Q q) {
this.q = q;
new Thread(this, "Consumer").start();
}
public void run() {
while(true) {
q.get();
} } }
class ProducerConsumer {
public static void main(String args[]) {
Q q = new Q();
new Producer(q);
new Consumer(q);
System.out.println("Press Control-C to stop.");
} }
```

Output:

```
Put: 1
Got: 1
Put: 2
Got: 2
Put: 3
Got: 3
Put: 4
Got: 4
Put: 5
Got: 5
```